Postgres OnLine Journal: December 2007

An in-depth Exploration of the PostgreSQL Open Source Database

Table Of Contents

A Product of Paragon Corporation

PostgreSQL The Road Behind and Ahead

Welcome to PostgreSQL OnLine Journal.

PostgreSQL is an extremely rich object relational database system and has a regal lineage that dates back almost to the beginning of the existence of relational databases.

If we were to look at the family tree of PostgreSQL it would look something like this

```
(Ingres, System-R)
          Postgres
              Illustra
                 Informix
                     IBM Informix
              Postgres95
                 PostgreSQL
```

In fact PostgreSQL is a cousin of the databases Sybase and Microsoft SQL Server because the people that started Sybase came from UC Berkeley and worked on the Ingres and/or Postgres projects with Michael Stonebraker. Later on the source code of Sybase SQL Server was later licensed to Microsoft to produce Microsoft SQL Server.

Here is an interesting diagram done by Oleg Bartunov that shows the various relational database pedigrees.

The main focus of this journal is to educate users and potential users about the numerous capabilities and uses of this powerful database management system.

Over the years we have watched PostgreSQL grow and reach a wider audience. Each day brings newer features, more stability, more environments supported and more Off-the-Shelf (OTS) applications that support this DBMS.

This Journal is a bit of a literary experiment for us. Technology is very fast-paced and we find that most of the new information we ingest these days comes via fast-paced sources such as Blogs and Magazine/ Periodical channels. I like the free form of the blog structure and ability to comment, but I also appreciate the more disciplined, carefully categorized, walk away with a booklet format of the Periodical. Our hope is to combine these two literary instruments into a *blogo-periodical* that has 2 faces:

- the face of a blog
- the face of an online magazine

.

Since this is what we call a *blogo-periodical* rather than a plain blog, we shall continually make edits to prior entries that are within the span of our editing issue in progress. So you may find if you are viewing it as a blog, that entries you have already read suddenly change.

After we complete each issue, we hope to provide each journal issue as a downloadable PDF magazine/ periodical. Issues in progress or completed will always be available as html ebooks.

In each issue of this journal, we hope to cover the following areas

1. **Editor's Note** - We will provide general comments we would like.
2. **What's New and Upcoming In PostgreSQL** - Will outline items targeted for next release as well as new features in current release.
3. **PostgreSQL Q&A** - Common questions and answers we have curled from newsgroups as well as user submitted questions
4. **Basics** - Articles that pose a basic problem or explain something
5. **PL Programming** - Using PL languages in PostgreSQL - e.g. Plpgsql, PLR, PlPython, PLPHP, PLPerl, PLPerlU, PLProxy etc.
6. **PostgreSQL Contribs Spotlight** - Using PostgreSQL contribs or advanced features - e.g. PostGIS, PgCrypto, PgSphere, TSearch etc.
7. **Application Development** - Using PostgreSQL in an Application e.g. with PHP, Python, Perl, Java, Ruby, .NET, MS Access, OpenOffice etc.
8. **Product Showcase** - a commercial or opensource product for managing PostgreSQL databases or that supports PostgreSQL as a backend

Back to Table Of Contents

PostgreSQL 8.3 is just around the Corner

PostgreSQL 8.3 is currently in Beta 4 and promises to offer some whoppingly neat features. First before we go over the new features we are excited about in this upcoming release, we'd like to briefly cover what was added in past releases.

The big 8.0 Highlights

1. The biggest feature in 8.0 was native support for Windows
2. Dollar quoting syntax for stored functions which made it a lot easier to write stored functions since instead of having to escape quotes you could use the $something$ body of function $something$ delimiter approach.
3. Next favorite was - Tablespaces so that people no longer needed to resort to messy symlinks to simulate this needed feature
4. Save points, and improved buffer management.

8.1 Highlights

1. Introduced Bitmap indexes which allowed for ability to use multiple indexes on a table simultaneously and less need for compound indexes.
2. Auto vacuuming. Auto Vacuuming was a huge benefit that all could appreciate. It allowed for automatic cleaning of dead space without human or scheduled intervention.
3. Also introduced in this release was improved shared locking and two phase commit
4. change in security - introducing login roles and group roles plus prevention of dropping roles that owned objects.
5. Constraint exclusion which improved speed of inherited tables thus improving table partitioning strategies.

8.2 Highlights

1. Query optimization improvements
2. 8.2 introduced multi row valued lists insert syntax (example here) similar to what MySQL has . As a side note, SQL Server 2008 will introduce a similar feature as row constructors.
3. Improved indexed creation that no longer required blocking concurrent insert, create, delete.
4. Ability to remove table inheritance from a child table without having to rebuild it.
5. Aggregates that can take multiple inputs and SQL:2003 statistical functions
6. Introduction of Fill Factor for tables similar to Microsoft SQL Server's Fill Factor functionality

8.3 upcoming Highlights

8.3 has numerous highlights just as previous versions, but we shall focus on our favorite ones.

1. Support Security Service Provider Interface (SSPI) for authentication on Windows - which presumably will allow PostgreSQL databases to enjoy the same single signon you get with Microsoft SQL Server 7-2005.
2. GSSAPI with Kerberos authentication as a new and improved authentication scheme for single signon
3. Numerous performance improvements - too many to itemize - check out Stefan Kaltenbrunner's

8.3 vs. 8.2 simple benchmark

4. The new QUERY functionality in plpgsql which offers a simpler way of returning result sets
5. Scrollable cursors in PLPgSQL
6. Improved shared buffers on windows
7. TSearch - Full Text Search is now integrated into PostgreSQL instead of being a contrib module
8. Support for SQL/XML and new XML datatype
9. ENUM datatype
10. New add-on feature to PgAdmin III - a PL debugger most compliments of EnterpriseDB

Back to Table Of Contents

Converting from Unix Timestamp to PostgreSQL Timestamp or Date ***Beginner***

A lot of databases structures people setup seem to store dates using the Unix Timestamp format (AKA Unix Epoch). The Unix Timestamp format in short is the number of seconds elapse since january 1, 1970. The PostgreSQL timestamp is a date time format (for those coming from SQL Server and MySQL) and of course is much richer than Unix Timestamp.

Question: *How do you convert this goofy seconds elapsed since January 1, 1970 to a real date?*

Answer:

PostgreSQL has lots of nice date time functions to perform these great feats. Lets say someone handed us a unix timestamp of the form 1195374767. We can convert it to a real date time simply by doing this: `SELECT TIMESTAMP 'epoch' + 1195374767 * INTERVAL '1 second'`.

Now lets say we had a table of said goofy numbers and we prefer real date structures. We can create a view that shields us from this mess by doing the following

```
CREATE OR REPLACE VIEW vwno_longer_goofy AS
SELECT goof.id, goof.stupid_timestamp,
      TIMESTAMP 'epoch' + goof.stupid_timestamp * INTERVAL '1 second' as ah_real_datetime
FROM goof
```

Back to Table Of Contents

Using Distinct ON to return newest order for each customer *Intermediate*

Question

If you have say a set of orders from customers and for each customer you wanted to return the details of the last order for each customer, how would you do it?

Answer

In databases that don't support the handy SQL DISTINCT ON or equivalent clause, you'd have to resort to doing subselects and MAXes as we described in SQL Cheat Sheet: Query By Example - Part 2. If you are in PostgreSQL however you can use the much simpler to write DISTINCT ON construct - like so

```
SELECT DISTINCT ON (c.customer_id)
       c.customer_id, c.customer_name, o.order_date, o.order_amount, o.order_id
      FROM customers c LEFT JOIN orders O ON c.customer_id = o.customer_id
      ORDER BY c.customer_id, o.order_date DESC, o.order_id DESC;
```

Back to Table Of Contents

How to create an index based on a function *Intermediate*

It is often handy to create indexes that are based on calculations of a function. One reason is that instead of storing the calculated value in the table along with the actual value, you save a bit on table scan speed since your row is thinner and also saves some disk space. It helps search speed if its a common function search.

Case in point, PostgreSQL is case sensitive so in order to do a simple search you will often resort to using upper or ILIKE. In those cases its useful to have an index on upper or lower cased text. Here is an example.

```
CREATE INDEX mem_name_idx
  ON member
  USING btree
  (upper(last_name), upper(first_name));
```

Here is another example taken from PostGIS land. Often times you provide your data in various transformation, but for space savings and row seek reasons, you want to only transform your data to the less used projections as needed. One way to do this is to create functional indexes on the commonly used transformations and create views or just write raw SQL that uses these alternative transformations.

```
   CREATE INDEX parcels_idx_geom_nadm
       ON parcels USING gist(ST_Transform(the_geom, 26986))
```

So now when I do a select like this that lists all buildings within 100 meters of my NAD 83 MA Meter State Plane point of interest:

```
    SELECT bldg_name, ST_Transform(the_geom,26986) As newgeom
        FROM buildings
        WHERE ST_DWithin(ST_Transform(the_geom, 26986) ,ST_GeomFromText('POINT(235675.754215375 894022.495855985)', 26986),
100)
```

it will use indexes

Back to Table Of Contents

The Anatomy of a PostgreSQL - Part 1 *Beginner*

In the next couple of sections we will outline the various things one will find in a PostgreSQL database. Many of these exist in other DBMS systems, but some of these are quite unique to PostgreSQL.

Exploring PostgreSQL with PgAdmin III

PgAdmin III is the Administrative console that comes packaged with PostgreSQL. It works equally well on most OSes - Linux, Unix, Windows, MacOS and any OS supported by WsWidgets. It is an extremely nice and capable management tool. PostgreSQL server comes packaged with this, but if you want to install this on a computer that doesn't have PostgreSQL server installed or you want the bleeding edge version or latest version, I suggest downloading from PgAdmin Site: http://www.pgadmin.org/download/. We will be exploring PostgreSQL with the newest stable releaseof PgAdmin III - 1.8.

When you first launch PgAdmin III and register your postgres server, you may be amazed at the number of things shown. In fact what is shown may not be all the objects that exist in PostgreSQL. PgAdmin III 1.8 and above hides a lot of things by default. For this exercise we will turn these settings on so we can see these objects and explore them.

To do so do the following

1. Open up PgAdminIII
2. Register a Postgres server if you haven't already by going to File->Add Server
3. Next go to File->Options and click on the *Display* tab
4. Check everything you see including the *Show System Objects in the treeview?*
5. Right mouse click on your registered server in the tree, click disconnect and then click connect

The Anatomy Lesson Begins

When you expand the Server tree, you will be first confronted with 4 groups of objects. As outlined below:

- **Databases** - Yap these are databases.
- **Tablespaces** - These define physical locations where stuff is kept. For people coming from a Microsoft SQL Server background (as I am), this is analogous to File Groups.
- **Group Roles** - As the name suggests these represent security groups.
- **Login Roles** - As the name suggests these represent logins.

In the next couple of sections, we will explore these areas a little deeper.

Databases

The first thing you will notice is that there are 3 system databases (databases you did not create) and they are *postgres*, *template0*, *template1*. These are outlined below

- **postgres** - database to hold system wide information. If you install PgAgent job scheduling agent - a tool we will cover in a later excerpt, this is usually installed as a *schema* in the Postgres

database.
- **template0, template1** - these are template databases used as boiler plates for new databases. For people coming from a Microsoft SQL Server background, template1 is analogous to the *model* database in SQL Server. In fact you can use any database you create as template for new databases.

Question: Why the heck are their 2 template databases?
Answer:
template1 is the default template used for new databases. Most people will use template1 as a template for their databases or create more derivative templates. template0 is basically a pristine template unadulterated by any thing except the core postgres stuff. In fact you can not change template0, but you can change template1.

Tablespaces

Tablespaces as I mentioned, represent physical locations on disk where things reside. There are 2 tablespaces installed by default:

- **pg_default** - This is the default tablespace where all user-defined objects are stored
- **pg_global** - This is the tablespace used to store system objects.

If you look at the location property of these 2 tablespaces, you will see nothing there. That is because these are always stored in the same location as where you initialized your PostgreSQL database cluster. Tablespaces that are user created on the other hand, can be stored anywhere on any disk and these you will see location information for.

In general there is rarely a reason to create new tablespaces and such unless you are creating a system with massive numbers of users, databases, and intensive queries. Tablespaces gives you the flexibility to leverage OS space in interesting ways - e.g. fast disks for commonly used tables, ability to have multiple disks in different RAID configurations for maximum seek performance, recoverability, or stability etc. Finding Optimum tables placement in 2 tablespace situation by Hubert Lubaczewski is particularly interesting. Also check out Robert Treat's tablespace configuration variable tweaking tips.

There are a couple of facts I would like to close with on the topic of tablespaces.

1. Databases are homed in a tablespace. Basically you may say a database resides in X tablespace, but not all tables of a database need to be stored in X tablespace.
2. The tablespace you denote for a database is the default location where objects of that database are stored. Also all objects that are not tables and indexes (e.g. functions, views, etc.) are stored in X tablespace and when creating new tables or indexes, they too will be stored in X tablespace unless otherwise noted.
3. As a corrollary to the above, you can only specifically designate a location for tables, indexes, and a database.

Group Roles and Login Roles

Prior to PostgreSQL 8.1, there existed Users and Groups, in 8.1 these were deprecated and replaced with Roles in order to be more ANSI compliant. This is actually a simplification of the security model. For more details check out the Chapter 18. Database Roles and Privileges

.

I'll summarize a few key facts about Group Roles and Login Roles

1. In PgAdmin III - Group Roles and Login Roles look like two different kinds of objects. In actuality

a Login Role is really a subclass of a group role so to speak. It is a role that has login rights.

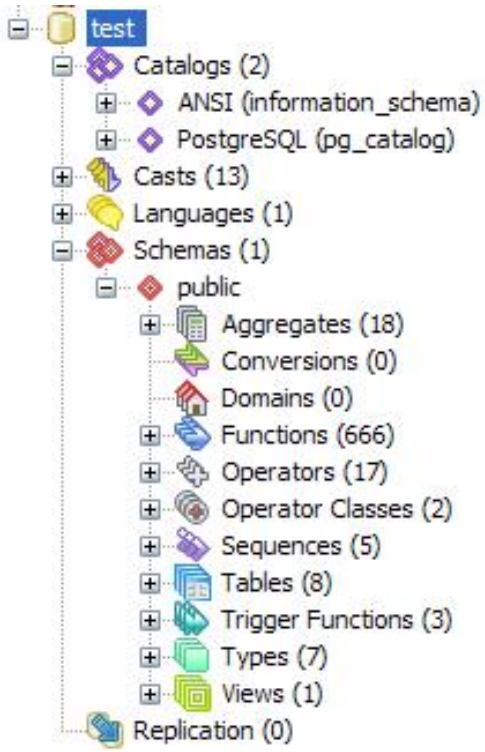
2. PostgreSQL has a feature which is a little different than some other databases, and that is that a role need not inherit rights from roles it is a member of.

   If you are not aware of this, it can bite you. For example if you create a login role and have it set to not inherit rights from its parent roles, you may be surprised to find out that even though the roles it is a member of has rights to certain tables, the login role does not. In order to use those rights, the logged in user needs to do `SET ROLE somerolename`.

   Now one may ask why would I ever need this feature? One reason that comes to mind is for debugging - you may want to create a user that is a member of several roles, and you want to test what rights each role has by constantly setting roles etc. Another reason is for security reasons - you may want to create a role that acts like an Application role and regardless of what login a user is logged into, you only want that user to have rights dictated by the security policy you set forth for that application. So within the application you could have logic that sets the role to that of the application, but allow the user to login with their standard login role.
3. Roles can be nested. You may have a role that is a member of yet another role, but you are prevented by the system from creating circular dependency roles -*Roles that are members of other roles that those roles are a member of these roles*
4. A role can be a member of multiple roles
5. LOGIN, SUPERUSER, CREATEDB, and CREATEROLE are not inheritable, but can be accessed by doing a SET ROLE call. Similar to in unix/linux when you do *sudo*

Coming Next Database Objects

In the next issue of this journal, we will go over database objects. In fact there are tons of these. I will leave you with a snapshot to wet your appetite.



Back to Table Of Contents

How does CLUSTER ON improve index performance *Intermediate*

CLUSTER Basics

One of the features in PostgreSQL designed to enhance index performance is the use of a clustered index. For people coming from MS SQL Server shops, this may look familiar to you and actually serves the same purpose, but is implemented differently and this implementation distinction is very important to understand and be aware of. In PostgreSQL 8.3 the preferred syntax of how you cluster has changed. For details check out 8.3 CLUSTER 8.2 CLUSTER 8.0 CLUSTER. A lot of what I'm going to say is somewhat of a regurgitation of the docs, but in slightly different words.

First in short - clustering on an index forces the physical ordering of the data to be the same as the index order of the index chosen. Since you can have only one physical order of a table, you can have only one clustered index per table and should carefully pick which index you will use to cluster on or if you even want to cluster. Unlike Microsoft SQL Server, clustering on an index in PostgreSQL does not maintain that order. You have to reapply the CLUSTER process to maintain the order. Clustering helps by reducing page seeks. Once an index search is done and found, pulling out the data on the same page is vastly faster since once you find the start point all successive data nearby is easy picking.

As a corrollary to the above, it doesn't help too much for non-range queries. E.g. if you have dummy ids for records and you are just doing single record select queries, clustering is fairly useless to you. It is only really useful if you are doing range queries like between date ranges or spatial ranges or queries where the neighboring data to an index match is likely to be pulled. For example if you have an order items table, then clustering on a compound index such as order_id,order_item_id may prove useful since neighboring data is something you likely want to pull for range and summations.

Now lets see how we create a clustered index and then talk about the pros and gotchas

```
--First we create the index
CREATE INDEX member_name_idx
  ON member
  USING btree
  (upper(last_name), upper(first_name));

ALTER TABLE member CLUSTER ON member_name_idx;
```

Once a clustered index is created to force a recluster, you simply do this

```
CLUSTER member;
```

To force a cluster on all tables that have clustered indexes, you do this

```
CLUSTER
```

What is FillFactor and how does it affect clustering?

Again those coming from Microsoft SQL Server will recognize FILLFACTOR syntax. IBM Informix also has a FILLFACTOR syntax that serves the same purpose as the SQL Server and PostgreSQL ones. For more

details here PostgreSQL docs: Create Index. FillFactor basically creates page gaps in an index page. So a Fill Factor of 80 means leave 20% of an index page empty for updates and inserts to the index so minimal reshuffering of existing data needs to happen as new records are added or indexed fields are updated in the system. This is incorporated into the index creation statement.

```
CREATE INDEX member_name_idx
  ON member
  USING btree
  (upper(last_name), upper(first_name))
  WITH (FILLFACTOR=80);
ALTER TABLE member CLUSTER ON member_name_idx;

-- note we do this to update the planner statistics information.
Its important since this information helps the planner at selecting indexes and scan approach.
ANALYZE member;
```

After an index is created on a table, this information is then used in several scenarios

- As stated earlier, as new records are added or indexed fields are updated, indexed values are shuffled into the empty slots and when new pages need to be created, they are created with the specified amount of space left blank.
- During vacuuming, reindexing again the specified amount of space is left blank
- During CLUSTERING, the clustering process tries to leave records where they are and uses the empty space to shuffle in the new data.

Why should you care?

First for fairly static tables such as large lookup tables, that rarely change or when they change are bulk changes, there is little point in leaving blank space in pages. It takes up disk space and causes Postgres to scan thru useless air. In these cases - you basically want to set your FillFactor high to like 99.

Then there are issues of how data is inserted, if you have only one index and new data usually resides at the end of the index and the indexed field are rarely updated, again having a low fill factor is probably not terribly useful even if the data is updated often. You'll never be using that free space so why have it.

For fairly updated data that changes such that you are randomly adding 10% new data per week or so in middle of page, then a fill factor of say 90 is the general rule of thumb.

Cluster approach benefits and Gotchas

The approach PostgreSQL has taken to cluster means that unlike the SQL Server approach, there is no additional penalty during transactions of having a clustered index. It is simply used to physically order the data and all new data goes to the end of the table. In the SQL Server approach all non-clustered indexes are keyed by the clustered index, which means any change to a clustered field requires rebuilding of the index records for the other indexes and also any insert or update may require some amount of physical shuffling. There are also other consequences with how the planner uses this information that are too detailed to get into.

The Bad and the Ugly

The bad is that since there is no additional overhead aside from the usual index key creation during table inserts and updates, you need to schedule reclustering to maintain your fine order and the clustering causes a table lock. The annoying locking hopefully will be improved in later versions. Scheduling a cluster can be done with a Cron Job or the more OS agnostic PgAgent approach. In another issue, we'll cover how to use PgAgent for backup and other scheduling maintenance tasks such as this.

Back to Table Of Contents

Language Architecture in PostgreSQL *Intermediate*

Perhaps one of the most unique and exciting things that makes PostgreSQL stand out from other database systems, are the numerous choices of languages one can use to create database functions, triggers and define new aggregate functions with. Not only can you use various languages to write your database stored functions with, but often times the code you write lives right in the database. You have no idea how cool this is until you see it in action.

The other interesting thing about the PostgreSQL language architecture is the relative ease with which new languages can be incorporated in the system.

Native Languages of PostgreSQL

There are 3 languages that come packaged with PostgreSQL (2 non-PL ones are installed automatically and not even listed as languages (C and SQL) in the languages section of a db). The defacto PL/PgSQL procedural language is available for install in all PostgreSQL distributions, but need not be installed in a db by default .

1. *C Extern* which allows for binding C libraries as functions. C Extern is similar to the way languages like MySQL bind C libraries for use in DB or the way SQL Server 2005+ binds .NET assemblies as functions in SQL Server.
2. *SQL* - this is a non-procedural language. It allows one to write parameterized db stored functions with plain SQL, but lacks procedural logic constructs such as IF, FOR, WHILE and so forth. It is basically a macro substitution language. Functions written in this way are basically in-lined in with the queries they are used (except in case of STABLE, IMMUTABLE defined in which case cached results are often used) in so they are more easily optimizable than functions written in other languages. NOTE: that MySQL 5+ also has a Procedural language called SQL, but the MySQL SQL language is a procedural language more in line with PostgreSQL *pl/pgsql* and closer in syntax to DB2's SQL PL. I'll also note that DB2 has a concept of INLINE SQL PL which is kind of like PostgreSQL sql language, although a bit more powerful.
3. *PL/PgSQL* - this is PostgreSQL defacto Procedural Language. It is not always installed by default in a database but the language handler is always available for installation. The equivalent but slightly different in syntax in other systems would be Transact SQL in SQL Server/Sybase, PL/SQL in Oracle, SQL in MySQL5+, and SQL PL in DB2.

The PL languages

Aside from PL/pgSQL there are numerous other procedural languages that one can use to create database stored functions and triggers. Some of these languages are fairly stable and even more are experimental. Some are only supported on Unix/Linux, but many are supported on Unix/Linux/MacOS/windows. In any case there are 3 key components needed before you can start using a new language:

1. The environment for the language - e.g. PHP, Perl, Python, Ruby, Java, R etc. interpreter and libraries installed on the PostgreSQL server box
2. The compiled call handler function - this is a C-compiled function that does the transfer between the PostgreSQL environment and the language environment.
3. The language registered in the database you wish to use it in.

Registering a language in a Database

For *pl/pgsql* items 1 and 2 are already done if you have a working PostgreSQL install. In order to accomplish item 3, you may need to do the following from psql or PgAdmin III query window.

```
CREATE TRUSTED PROCEDURAL LANGUAGE 'plpgsql'
 HANDLER plpgsql_call_handler
 VALIDATOR plpgsql_validator;
```

Alternatively you can run *createlang plpgsql somedb* from commandline. Note createlang is a command line program that is located in the **bin** folder of your PostgreSQL install.

To see a list of procedural languages that you already have call handlers registered for in PostgreSQL. These are the languages you can register in your specific database - do a

```
SELECT * FROM pg_catalog.pg_pltemplate
```

A Flavor of the Procedural Languages (PLs)

In this section, we'll show a brief sampling of what functions look like written in various PLs. These are not to suggest they are the only ones that exist. For these examples, I'm going to use the $ quoting syntax introduced in PostgreSQL 8.0 which allows for not having to escape out single quotes.

SQL - the not PL language

For basic CRUD stuff,selects and simple functions, nothing hits the spot like just plain old SQL. Since this is such a common choice and often the best choice - here are 3 examples.

```
CREATE OR REPLACE FUNCTION cp_simpleupdate(thekey integer, thevalue varchar(50))
  RETURNS void AS
        $BODY$
                UPDATE testtable SET test_stuff = $2 WHERE test_id = $1
        $BODY$
  LANGUAGE 'sql' VOLATILE;
```

```
--Example use
SELECT cp_simpleupdate(1, 'set to this');
```

```
--Here is a simple example to simulate the MySQL 5.0 function
CREATE OR REPLACE FUNCTION from_unixtime(unixts integer)
  RETURNS timestamp without time zone AS
$BODY$SELECT CAST('epoch' As timestamp) + ($1 * INTERVAL '1 second') $BODY$
  LANGUAGE 'sql' IMMUTABLE;

  --Example use
  SELECT from_unixtime(1134657687);
  SELECT from_unixtime(tbl.fromsomefield) FROM tbl;
```

```
CREATE OR REPLACE FUNCTION cp_test(subject varchar)
  RETURNS SETOF testtable AS
  $BODY$
        SELECT * FROM testtable where test_stuff LIKE $1;
        $BODY$
  LANGUAGE 'sql' VOLATILE;
  --Example use
  SELECT * FROM cp_test('%stuff%');
```

```
  CREATE OR REPLACE FUNCTION cp_testusingoutparams(subject varchar, out test_id int, out test_stuff varchar)
  RETURNS SETOF record AS
$BODY$
        SELECT test_id, test_stuff FROM testtable where test_stuff LIKE $1;
$BODY$
```

```
  LANGUAGE 'sql' VOLATILE;

--Example use - Note the subtle difference - the second syntax with out parameters is newer
-- It allows you to get around the messy issue of when you are returning a record type
--That a record type has no specific type.
SELECT * FROM cp_usingoutparams('%stuff%');
```

For details on using out parameters, check out Robert Treat's out parameter sql & plpgsql examples

PLPGSQL - a real PL Language

For more complex logic and massaging of results before sending back. You need something more powerful than standard SQL. Below are some examples using PLPGSQL.

```
CREATE OR REPLACE FUNCTION cp_harderupdate(thekey integer, thevalue varchar)
RETURNS void AS
$BODY$
BEGIN
        IF EXISTS(SELECT test_id FROM testtable WHERE test_id = thekey) THEN
                UPDATE testtable SET test_stuff = thevalue WHERE test_id = thekey;
        ELSE
                INSERT INTO testtable(test_id, test_stuff) VALUES(thekey, thevalue);
        END IF;
     RETURN;
END;
$BODY$
LANGUAGE 'plpgsql' VOLATILE;

--Example use
SELECT cp_harderupdate(1, 'this is more stuff');
```

Using PL/Perl

```
CREATE OR REPLACE FUNCTION get_neworders() RETURNS SETOF orders AS $$
    my $rv = spi_exec_query('select * from orders where processed IS NULL;');
    my $status = $rv->{status};
    my $nrows = $rv->{processed};
    foreach my $rn (0 .. $nrows - 1) {
        my $row = $rv->{rows}[$rn];
        return_next($row);
    }
    return undef;
$$ LANGUAGE plperl;
```

Using PL/R a language and environment for statistics

One of my favorite PL languages to program is PL/R. The reason for this is that the R statistical environment is such a rich environment for doing statistical processing. It now is also supported on windows as well as Mac and Linux.

To learn more about R and installing PL/R. Check out our Boston GIS article *PLR Part 1: Up and Running with PL/R (PLR) in PostgreSQL: An almost Idiot's Guide*

Below is the classic median aggregate function in R. It uses the native median function in the R environment to create a PostgreSQL aggregate median function

```
CREATE or REPLACE FUNCTION r_median(_float8)
        returns float as $BODY$ median(arg1) $BODY$ language 'plr';

CREATE AGGREGATE median (
  sfunc = plr_array_accum,
  basetype = float8,
  stype = _float8,
  finalfunc = r_median
);
```

```
--Example use
SELECT median(age) As themedian_age, period_year
        FROM crimestats GROUP BY period_year ORDER BY period_year;
```

We will be covering PLR in greater detail in another article.

Back to Table Of Contents

PostGIS for geospatial analysis and mapping *Intermediate*

In later issues we'll be covering other PostgreSQL contribs. We would like to start our first issue with introducing, PostGIS, one of our favorite PostgreSQL contribs. PostGIS spatially enables PostgreSQL in an OpenGeospatial Consortium (OGC) compliant way. PostGIS was one reason we started using PostgreSQL way back in 2001 when Refractions released the first version of PostGIS with the objective of providing affordable basic OGC Compliant spatial functionality to rival the very expensive commercial offerings. There is perhaps nothing more powerful in the geospatial world than the succinct expressiveness of SQL married with spatial operators and functions. Together they allow you to manipulate and analyze space with a single sentence. For details on using Postgis and why you would want to, check out the following links

- Introduction to PostGIS
- Webb Sprague's talk on PostGIS (Geographic Databases)
- PostGIS site
- Almost idiot's guide to PostGIS
- PostGIS Cheatsheet

Just as PostgreSQL has grown over the years, so too has PostGIS and the whole FOSS4G ecosystem. PostGIS has benefited from both the FOSS4G and PostgreSQL growths. On the PostgreSQL, improvements such as improved GIST indexing, bitmap indexes etc and on the FOSS4G side dependency projects such as Geos and Proj4, and JTS, as well as more tools and applications being built on top of it.

In 2001 only UMN Mapserver was available to display PostGIS spatial data. As time has passed, UMN Mapserver has grown, and other Mapping software both Commercial and Open Source have come on board that can utilize PostGIS spatial data directly. On the FOSS side there are many, some being UMN Mapserver, GRASS, uDig, QGIS, GDAL/OGR, FeatureServer, GeoServer, SharpMap, ZigGIS for ArcGIS integration, and on the commercial side you have CadCorp SIS, Manifold, MapDotNet, Safe FME Data Interoperability and ETL tools.

In terms of spatial databases, PostGIS is the most capable open source spatial database extender. While MySQL does have some spatial capabilities, its spatial capabilities are extremely limited particularly in the selectivity of the spatial relational functions which are all MBR only, ability to create spatial indexes on non-MyISAM stores, and lack a lot of the OGC compliant functions such as Intersection, Buffering even in its 5.1 product. For details on this check the MySQL 5.1 docs - Spatial Extensions.

When compared with commercial spatial databases, PostGIS has most of the core functions you will see in the commercial databases such as Oracle Spatial, DB2 Spatial Blade, Informix Spatial Blade, has comparable speed, fewer deployment headaches, but lacks some of the advanced add-ons you will find, such as Oracle Spatial network topology model, Raster Support and Geodetic support. Often times the advanced spatial features are add-ons on top of the standard price of the database software.

Some will argue that for example Oracle provides Locator free of charge in their standard and XE versions, Oracle Locator has a limited set of spatial functions. Oracle's Locator is missing most of the core spatial analysis and geometric manipulation functions like centroid, buffering, intersection and spatial aggregate functions; granted it does sport geodetic functionality that PostGIS is currently

lacking. To use those non-locator features requires Oracle Spatial and Oracle Enterprise which would cost upwards of $60,000 per processor. Many have heard of SQL Server 2008 coming out and the new spatial features it will sport which will be available in both the express and the full version. One feature that SQL Server 2008 will have that PostGIS currently lacks is Geodetic support (the round world model so to speak). Aside from that SQL Server 2008 has a glarying omission from a current GIS perspective - and that is the ability to transform from one spatial reference system to another directly in the database and is Windows bound so not an option for anyone who needs or is thinking of cross-platform or in a Unix environment. SQL Server 2008 will probably come closest to PostGIS in terms of price / functionality. The express versions of the commercial offerings have many limitations in terms of size of database and usually limited to one processor use. For any reasonably sized deployment in terms of database size, processor utilization, replication, or ISP/Service Provider/Integrator this is not adequate and for any reasonably large deployment that is not receiving manna from heaven, some of the commercial offerings like Oracle Spatial, are not cost-sensible.

Note that in near future versions PostGIS is planning to have geodetic support and does provide basic network topology support via the PgRouting project and there are plans to incorporate network topology as part of PostGIS.

There is a rise in the use of mapping and geospatial analysis in the world and it is moving out of its GIS comfort zone to mingle more with other IT Infrastructure, General Sciences, and Engineering. Mapping and the whole Geospatial industry is not just a tool for GIS specialists anymore. A lot of this rise is driven by the rise of mapping mashups - things like Google Maps, Microsoft Virtual Earth, and Open data initiatives that are introducing new avenues of map sharing and spatial awareness. This new rise is what many refer to as NeoGeography. NeoGeography is still in its infancy; people are just getting over the excitement of seeing dots in their hometown, and are quickly moving into the next level - where more detailed questions are being asked about those dots and dots are no longer sufficient. We want to draw trails such as trail of hurricane destruction, avian bird flu, track our movement with GPS, draw boundaries and measure the densities of these based on some socio-ecological factor and we need to store all that user generated or tool generated information, and have all that transactional goodness, security and ability to query in an easy way that a relational database offers. This is the level where PostGIS and other spatial databases are most useful.

Back to Table Of Contents

Database Abstraction with Updateable Views *Advanced*

One of the annoying things about PostgreSQL unlike some other databases we have worked with is that simple views are not automatically updateable. There is some work involved to make views updateable. For simple views, this is annoying, but for more complex views it is a benefit to be able to control how things are updated. In a later version of PostgreSQL perhaps 8.4 or 8.5 this will be ratified and PostgreSQL will enjoy the same simplicity of creating simple updateable views currently offered by MySQL and SQL Server and other DBMSs, but still allow for defining how things should be updated for more complex views. For this exercise we are using PostgreSQL 8.2.5, but most of it should work for lower versions with slight modification.

For this exercise, we shall create a fairly complex updateable view to demonstrate how one goes about doing this.

Here is a scenario where being able to control how a view is updated comes in very handy.

We all know relational databases are great because they give you great mobility on how you slice and dice information. At times for data entry purposes, the good old simple flat file is just more user-friendly.

Problem: You are developing an inventory application for a molecular biology lab and they have the following requirements:

1. They want to keep track of how much of each supply they use for each project grant for funding purposes and report on that monthly or daily.
2. They want to keep track of how much of each supply they ordered, what they have left and their usage over time.
3. They however want data entry to be as simple as possible. They want a simple flat file structure to input data that has columns for each project usage and column for purchase quantity.

They have 2 projects going on. One on Multiple Sclerosis Research (MS) and one on Alzheimer's. Each is funded by different grants and for grant cost allocation purposes, they need to keep track of the supplies they use on each project.
How do you present a flat file inventory entry screen, but behind the scenes have a inventory and inventory transaction scheme so you can run period reports and aggregate summaries and have automatic totaling?
Possible Solution: One way to do it is with a crosstab summary view that is updateable. Views are incredibly useful abstraction tools. You do that in PostgreSQL by creating insert, update, and delete rules on your views. For our particular case, we will not be allowing deletion so we will not have a delete rule.

In our system we have 2 tables for simplicity. inventory and inventory_flow. I know we should have a project lookup table or in 8.3 possibly use an ENUM, but to make this short, we are skipping that.

```
CREATE TABLE inventory
(
  item_id serial NOT NULL,
  item_name varchar(100) NOT NULL,
  CONSTRAINT pk_inventory PRIMARY KEY (item_id),
  CONSTRAINT inventory_item_name_idx UNIQUE (item_name)
)
WITH (OIDS=FALSE);
```

```
CREATE TABLE inventory_flow
(
  inventory_flow_id serial NOT NULL,
  item_id integer NOT NULL,
  project varchar(100),
  num_used integer,
  num_ordered integer,
  action_date timestamp without time zone NOT NULL DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT pk_inventory_flow PRIMARY KEY (inventory_flow_id),
  CONSTRAINT fk_item_id FOREIGN KEY (item_id)
      REFERENCES inventory (item_id)
      ON UPDATE CASCADE ON DELETE RESTRICT
)
WITH (OIDS=FALSE);



CREATE VIEW vwinventorysummary As
        SELECT i.item_id, i.item_name,
SUM(CASE WHEN iu.project = 'Alzheimer''s'
               THEN iu.num_used ELSE 0 END) As total_num_used_altz,
           SUM(CASE WHEN iu.project = 'MS' THEN iu.num_used ELSE 0 END) As total_num_used_ms,
           CAST(NULL As integer) As add_num_used_altz,
           CAST(NULL As integer) As add_num_used_ms,
           CAST(NULL As integer) As add_num_ordered,
           SUM(COALESCE(iu.num_ordered,0)) - SUM(COALESCE(iu.num_used,0)) As num_remaining
        FROM inventory i LEFT JOIN inventory_flow iu ON i.item_id = iu.item_id
        GROUP BY i.item_id,  i.item_name;

        CREATE RULE updinventory AS
                ON UPDATE TO vwinventorysummary
                        DO INSTEAD (
                                UPDATE inventory
                                   SET item_name = NEW.item_name WHERE inventory.item_id = NEW.item_id;

                                INSERT INTO inventory_flow(item_id, project, num_used, num_ordered)
                                                SELECT NEW.item_id, 'Alzheimer''s', NEW.add_num_used_altz, 0
                                                WHERE NEW.add_num_used_altz IS NOT NULL;
                                INSERT INTO inventory_flow(item_id, project, num_used, num_ordered)
                                                SELECT NEW.item_id, 'MS', NEW.add_num_used_ms, 0
                                                WHERE NEW.add_num_used_ms IS NOT NULL;
                                INSERT INTO inventory_flow(item_id, project, num_used, num_ordered)
                                                SELECT NEW.item_id, 'Resupply', 0, NEW.
add_num_ordered
                                WHERE NEW.add_num_ordered IS NOT NULL;);

        CREATE RULE insinventory AS
                ON INSERT TO vwinventorysummary
                DO INSTEAD (
                        INSERT INTO inventory (item_name) VALUES (NEW.item_name);
                        INSERT INTO inventory_flow (item_id, project, num_used)
                            SELECT i.item_id AS new_itemid, 'Altzeimer''s', NEW.add_num_used_altz
                                        FROM inventory i
                                        WHERE i.item_name = NEW.item_name
                                           AND NEW.add_num_used_altz IS NOT NULL;

                        INSERT INTO inventory_flow (item_id, project, num_used)
                          SELECT i.item_id AS new_item_id, 'MS', NEW.add_num_used_ms
                                FROM inventory i
                                WHERE i.item_name = NEW.item_name AND
                                         NEW.add_num_used_ms IS NOT NULL;

                        INSERT INTO inventory_flow (item_id, project, num_ordered)
                             SELECT i.item_id AS new_item_id, 'Initial Supply', NEW.add_num_ordered
                                FROM inventory i
                                        WHERE i.item_name = NEW.item_name AND
                                          NEW.add_num_ordered IS NOT NULL;
                );
```

Now look at what happens when we insert and update our view

```
 --NOTE: here we are using the new multi-row valued insert feature introduced in 8.2
        INSERT INTO vwinventorysummary(item_name, add_num_ordered)
                VALUES ('Phenol (ml)', 1000), ('Cesium Chloride (g)', 20000),
                        ('Chloroform (ml)', 10000), ('DNA Ligase (ml)', 100);

        UPDATE vwinventorysummary
                SET add_num_used_ms = 5, add_num_used_altz = 6 WHERE item_name = 'Cesium Chloride (g)';
        UPDATE vwinventorysummary SET add_num_used_ms = 2 WHERE item_name = 'Phenol (ml)';
        UPDATE vwinventorysummary SET item_name = 'CSCL (g)' WHERE item_name = 'Cesium Chloride (g)';
```

The slick thing about this is if you were to create a linked table in something like say Microsoft Access and designated item_id as the primary key, then the user could simply open up the table and update as normally and behind the scenes the rules would be working to do the right thing.

Back to Table Of Contents

Serendipity Blogging Software

Choosing Blogging Software

When we started blogging, we had several criteria for the blogging software we would use.

- Easy to install
- Easy to use
- Couldn't be a service and the database structure needed to be fairly easy to understand because we needed to mesh it seamlessly with the rest of our site.
- Had to support PostgreSQL
- Preferably open source and based on technology we understood - that meant either ASP.NET or PHP
- As far as code goes we are pretty finicky about those things and for PHP we prefer the Smarty Templating system and PHP ADODB over other PHP paradigms. Part of that, not to insult others, was that it was the first approach we found that worked really well for us, so we stuck with it.

We immediately dismissed wordpress because it was MySQL centric, Blogger etc services were out the door as well. There were not that many blogging applications in .NET and most were very SQL Server centric.

We noticed other PostgreSQL bloggers use predominantly Serendipity, so we thought we'd give it a try.

Serendipity met all our requirements except for the PHP ADODB part. It has a database abstraction layer, but it appears to be a custom one. This we could live with. Below are the features we really liked about it.

- Works on both Linux and Windows - IIS and Apache
- Easy installation. Install process was literally 10 minutes or less
- Supports PostgreSQL, MySQL, and SQLite
- Underlying database structure was sane
- Its underlying templating system is PHP Smarty-Based
- At least it had a database abstraction layer. For .NET development we've built our own because all the ones out there went too far in their abstraction to the point of being counter-intuitive and .NET doesn't really have a pre-packaged database abstraction layer to speak of. So this particular choice of decisions was one we could accept.
- Fairly intuitive.
- Plug-ins galore - in fact most of our time was spent figuring out which plug-ins we wanted to use.
- Ability to assign multiple categories to a blog post and threaded categories
- Fairly straight-forward theming system

Choosing Plugins

There are some plugins enabled by default, but can't remember which ones. For the most part they are the common ones people would choose if they chose them. These get you pretty far at least to use the software before you realize hey there is other stuff you can turn on or off. Below are some of the ones we found as must haves or things that should think about changing.

Serendipity has plugins broken up into event plugins and side bar plugins. Side bar plugins can be drag and dropped between the left right middle areas, which is a nice convenience. Event plugins are triggered based on Serendipity system events such as blog posts or comment posts and some aren't really events so to speak but aren't side bar plugins either so they show under events.

Event Plug-ins

1. To WYSIWYG or Not? Serendipity's WYSIWYG setting is set at the user level which is nice since some people like it and some don't and if you have a group blog this is very useful. Personally we don't care for WYSIWYG especially for a site that shows coding snippets. This is not to say that WYSIWYG in Serendipity is not adequate for a lot of people. Just not for us. We never use the WYSIWYG in Visual Studio either. I blame being brought up writing papers in LATEX for this frustration with WYSIWYG.
2. Markup: NL2BR - If you are going to be writing your own HTML turn this off for blog body. It screws up your nice formatting since it will literally turn each newline into a break when presented. Should probably always have this turned on for comments otherwise people writing out carefully thought out comments will be frustrated when their paragraphs are squashed.
3. [S]erendipity [P]lugin [A]ccess [R]epository [T]ool [A]nd [C]ustomization/[U]nification [S]ystem (SPARTACUS) - this is a plug-in that allows you to connect to the Serendipity plug-in and update your plug-in repository - kind of like a **YUM** for Serendipity.
4. Announce Entries - this is a plug-in that does an XML-RPC ping post to places like technorati, google, ping-o-matic. You can enable and disable which ones you want posted to by default when your entry is published. Within the entry screen, you can selectively uncheck and check them as well for that particular entry

Sidebar Plugins

We haven't played with these too much. The standard default calendar, category, and search were pretty much what we needed starting off. We liked the Wiki Finder and the links to publish to social bookmarking sites as a nice convenience.

Gripes

We also tried this on a virgin install of PostgreSQL 8.3 Beta 3 and it didn't work. Seems to be some logic in the DB layer of serendipity that uses LIKE instead of = against ids and the fact that PostgreSQL 8.3 has taken out a lot of the default CASTS. I think the serendipity code should be changed in this case since from a cursory glance, doesn't quite look right or efficient, but I'm sure there is a good reason they chose to do things that way.

Back to Table Of Contents

Reader Comments

PostgreSQL 8.3 is just around the Corner

Tony Caduto

Hi,
Just wanted to let you know that we have a FREE stand alone Debugger client for 8.3 available that does not require PG Admin or any other Admin program for that matter. So far from my testing it's far more stable than the one shipped with PG Admin III. Check it out at http://www.amsoftwaredesign.com

---

How does CLUSTER ON improve index performance

Robert Treat

I don't know exactly when CLUSTER was introduced, but it's been around since at least 6.5.

Also, if memory servers, there has been some discussion of clustering tables in the sql server sense; I think it has been reffered to as Index Ordered Tables on -hackers. If there is enough interest, it might end up included in 8.4. We'll see.

Leo

Thanks for the info. We'll correct our entry.

Language Architecture in PostgreSQL

David Fetter

No offense, but you are mistaken about SQL. With functions, it is Turing-complete.

Regina

David,

Which part are you referring to in your comment? We weren't trying to imply that SQL can not be Turing-complete, just that for certain expressions of logic it is not the best choice and a procedural language is better. If I am not mistaken I would say all procedural languages are Turing complete, but not all Turing complete languages are procedural in nature.

David Fetter

SQL is not a "macro substitution language" any more than C is. It can recurse, do many different kinds of control structures, etc. That you're used to using it in a declarative way does not imply that that's the only way it can be used.

I include here a brief example of a blindingly fast memoizing Fibonacci calculator written in Postgres's version of SQL.

```
CREATE TABLE fib_mem(
n numeric PRIMARY KEY,
fib_n numeric NOT NULL
);

CREATE OR REPLACE FUNCTION memoize_fib(n numeric, fib_n numeric)
RETURNS numeric
STRICT
LANGUAGE SQL
AS $$
INSERT INTO fib_mem VALUES ($1, $2);
SELECT $2;
$$;

CREATE OR REPLACE FUNCTION fib(numeric)
RETURNS numeric
LANGUAGE SQL
```

```
AS $$
SELECT COALESCE(
(SELECT fib_n FROM fib_mem WHERE n=$1),
memoize_fib(
$1,
CASE WHEN $1 < 2 THEN $1 ELSE fib($1-2) + fib($1-1) END
)
);
$$;
```

Regina
Interesting. When we mentioned macro substitution we were referring to how the planner optimizes the SQL functions. Perhaps our choice of terms was confusing.

Of course this is just a cursory observation on our part not from looking at the postgresql source code, but from looking at how the planner was utilizing indexes. Our guess is that with SQL functions the functions are inlined with the larger plan where it is used kind of like the way you can inline c functions. It seemed the same thing isn't done with the other languages you can write functions in.

For example I would bet if you did the same thing in plpgsql it would be slower than the above. Well maybe not for this case, but for other cases we have tried.

PostGIS for geospatial analysis and mapping

romi
It would be better if you can also mention GEOS and PROJ4 project references, because I think the spatial analysis function and projection support in PostGIS would also be limited when both are missing.

Regina
Very good point. I'll add that in :)

Trophaeum
This is exactly the article i needed to get my backside into learning postgis! thanks!

Database Abstraction with Updateable Views

joseph.randomnetworks.com

Postgres OnLine Journal
In the first part of this series, we covered PostgreSQL Server object features. In this part, we shall demo
the database and dissect the parts.

Here we see a snapshot of what a standard PostgreSQL database looks like from a PgAdmin interface.

Serendipity Blogging Software

Andreas Scherbaum
My own Serendipity blog runs on PostgreSQL, but for some modules i submitted patches and/or bug reports.

The entire SQL code is mysqlisch and the database layer does a hard job rewriting the stuff into something more standard SQL ... sometimes this fails.
As example, the "last google search" module once stopped working, without error and all. After some debugging i found out, that a required database update could not applied and was silently ignored.

PgSQL 8.3 beta4
Serendipity works like a charm on beta4 (didn't try with beta3)

Leo and Regina

Thanks for the note. We'll give it a try on the next 8.3 we install. I suppose it could be the version of Serendipity too. We had tested on 1.3 beta. Looks like you are running on 1.2.1.
